The MIRV SimpleScalar/PISA Compiler

Matthew Postiff, David Greene, Charles Lefurgy, Dave Helder and Trevor Mudge
{postiffm,greened,lefurgy,dhelder,tnm}@eecs.umich.edu
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122

Abstract

We introduce a new experimental C compiler in this report. The compiler, called MIRV, is designed to enable research that explores the interaction between the compiler and microarchitecture. This introductory paper makes comparisons between MIRV and GCC. We notice trends between the compilers and optimization levels across SPECint1995 and SPEC2000. Finally, we provide a set of SimpleScalar/PISA binaries to the research community. As we improve the compiler, we encourage architecture researchers to use these optimized binaries as reference programs for architecture research.

1. Introduction

The design of computers in general and microprocessors in particular has shown a steady increase in both performance and complexity. Advanced techniques such as pipelining and out-of-order execution have increased the design and verification effort required to create a viable product. To overcome some of these problems, hardware designers have been exploring ways to move functionality into the compiler. From RISC to current designs such as Intel's IA64, the compiler has played a greater role in simplifying the hardware while maintaining the current trend of performance improvement.

The MIRV compiler is designed to analyze trade-offs between compile-time and run-time knowledge of program behavior. MIRV enables research into this area in four ways. First, the compiler is built with a modular filter architecture. This allows the researcher to easily write optimizations and explore their placement in the phase ordering. Second, the retargetable code generator and low-level optimizer support both commercially available microprocessors and the popular SimpleScalar simulation environment. This allows both realistic performance evaluation as well as explorations into next-generation computer instruction set architecture. Third, MIRV provides an interface for program instrumentation and profile back-annotation. This allows studies into runtime behavior as well as profile-guided optimizations. Fourth, the compiler environment that we have developed around MIRV provides easy regression testing, debugging, and extraction of performance characteristics of both the compiler and the compiled code.

In this report we introduce MIRV and compare its performance to the GCC compiler. This report also introduces a package of SPEC binary executables which are compiled with GCC and MIRV at various optimization levels. The purpose of this document is to explain the compilation and simulation environment in which the binaries were produced and to summarize the performance differences between the compiled code. Several notable results are presented.

The organization of the rest of this paper is as follows. Section 2 describes the compilation environment that we used to generate the results shown. Similarly, Section 3 outlines the simula-

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
29 MAR 2000		**00-03-2000 to 00-03-2000**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The MIRV SimpleScalar/PISA Compiler	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Michigan,Computer Science and Engineering Division,Department of Electrical Engineering and Computer Science,Ann Arbor,MI,48109-2122**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**23**	

tion environment. Section 4 introduces the performance graphs shown in the appendices and Section 5 describes some interesting observations made from the performance graphs. We conclude with Section 6. The appendices contain detailed compilation and simulation results as well as provide additional detail on the optimizations that were performed during compilation.

2. Compilation Environment

We tested seven compiler configurations. The first is labeled ‘SSsup’ which is the SimpleScalar supplied binary, available at the SimpleScalar web site [2]. The next three configurations were compiled in our test environment with the GCC 2.7.2.3 port to the PISA instruction set. This tool is available from UC-Davis [7]. We also used a pre-release version of binutils 2.9.5 for the assembler and linker. These were slightly modified from sources at Cygnus [8]. The final three configurations were compiled with MIRV and used the same assembler and linker as the GCC builds.

The MIRV compiler implements the most common optimization passes. The exact order of application of the optimzation filters is given in Table 4 in Appendix A. For comparison, Appendix B contains the optimizations applied in the GCC compiler.

MIRV always applies register coalescing and graph coloring register allocation in the backend, regardless of the optimization level. The allocator is implemented with the standard graph coloring algorithm except that it does not implement live range splitting or rematerialization [3]. This means that it is not fair to compare GCC -O0 with mirv -O0 since GCC does not perform register allocation at the -O0 optimization level.

3. Simulation Environment

The SimpleScalar 3.0 sim-outorder simulator was used with default parameters [4]. Table 1 shows the relevant default parameter values. All simulations were performed in little-endian mode.

We used the SPEC95 integer benchmarks and several of the SPEC00 benchmarks [5, 6]. All benchmarks were run to completion on the data set indicated in the table; we modified the supplied input sets to allow the simulations to complete in a reasonable amount of time (about 100 million instructions). The benchmarks are described in Table 2 and the exact input sets are shown in Table 3.

4. SPEC Performance Graphs

The full set of graphs comparing MIRV to GCC can be found in Appendices C and D. These graphs show various metrics for each of the eight SPEC95 benchmarks and selected SPEC00 benchmarks. Table 6 explains each of the graphs and any special notes on how the data was gathered. For the SPEC95 benchmarks, we include the PISA binary supplied on the SimpleScalar website [2] as a comparison point. These benchmarks were compiled with the arguments “-O2 -funroll-loops”. There are no supplied binaries for SPEC00 benchmarks, so no information appears for those in our graphs. The full set of results is attached in Appendix F.

SimpleScalar parameter	Value
fetch queue size	4
fetch speed	1
decode, width	4
issue width	4 out-of-order, wrong-path issue included
commit width	4
RUU (window) size	16
LSQ	8
FUs	alu:4, mult:1, memport:2, fpalu:4, fpmult:1
branch prediction	2048-entry table of 2-bit counters, 4-way 512-set BTB, 3 cycle extra mispredict latency, non-speculative update, 8-entry return address stack
L1 D-cache	128-set, 4-way, 32-byte lines, LRU, 1-cycle hit, total of 16KB
L1 I-cache	512-set, direct-mapped 32-byte line, LRU, 1-cycle hit, total of 16KB
L2 unified cache	1024-set, 4-way, 64-byte line, 6-cycle hit, total of 256KB
memory latency	18 cycles for first chunk, 2 thereafter
memory width	8 bytes
Instruction TLB	16-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty
Data TLB	32-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty

Table 1. Simulation parameters for sim-outorder (the defaults).

Category	Benchmark	Description
SPECint95	compress	A in-memory version of the common UNIX utility.
	gcc	Based on the GNU C compiler version 2.5.3.
	go	An internationally ranked go-playing program.
	ijpeg	Image compression/decompression on in-memory images.
	li	Xlisp interpreter.
	m88ksim	A chip simulator for the Motorola 88100 microprocessor.
	perl	An interpreter for the Perl language.
	vortex	An object oriented database.
SPECfp2000	art	Recognizes objects in a thermal image using a neural network.
	equake	Simulation of seismic wave propagation in large basins.
SPECint2000	gzip	Data compression program that uses Lempel-Ziv coding (LZ77) as its compression algorithm.
	mcf	A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation.
	vortex	A single-user object-oriented database transaction benchmark which exercises a system kernel coded in integer C.
	vpr	Performs placement and routing in Field-Programmable Gate Arrays.

Table 2. Descriptions of the benchmarks used in this study.

The only anomalous behavior we observed during simulations was in the vortex benchmark, where we discovered that the SimpleScalar supplied binary had been compiled with the flag '-DOPTIMIZE'. The GCC and MIRV binaries that we initially built were not compiled with this flag because we did not know about it. The flag turns on various optimizations in the vortex code

Category	Benchmark	Input
SPECint95	compress	30000 q 2131
	gcc	regclass.i
	go	9 9 null.in
	ijpeg	specmun.ppm, -compression.quality 25, other args as in training run
	li	boyer.lsp (reference input)
	m88ksim	ctl.lit (train input)
	perl	jumble.pl < jumble.in, dictionary up to 'angeline' only
	vortex	250 parts and 1000 people, other variables scaled accordingly
SPECfp2000	art	-scanfile c756hel.in -trainfile1 a10.img -stride 2 -startx 134 -starty 220 -endx 139 -endy 225 -objects 1 (test input)
	equake	< inp.in (test input)
SPECint2000	gzip	input.compressed 1 (test input)
	mcf	inp.in (test input)
	vortex	250 parts and 1000 people, other variables scaled accordingly
	vpr	net.in arch.in place.in route.out -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8 (test input)

Table 3. Description of benchmark inputs.

itself (it is a preprocessor directive). We added '-DOPTIMIZE' to our simulations and the anomaly was solved.

5. Performance Observations

Several interesting observations can be made from the data shown in Appendices C and D. These observations could fall into several categories which are examined in the following subsections. It is important to keep in mind the simulator configuration shown in Table 1.

5.1 Comparing MIRV to GCC

GCC has no register allocation in -O0. MIRV has graph coloring allocation and register coalescing (simple copy propagation). Since GCC and MIRV unoptimized code is otherwise very similar, we can use these two bars to show an estimate of the importance of register allocation. For example, MIRV -O0 execution times are often 20% faster than GCC -O0 and sometimes much faster. This benefit is solely due to register allocation. MIRV-O1 and -O2 performs a little worse than GCC. This is borne out in the graphs on cycles and dynamic counts of instructions, memory references and branches. The dynamic instruction mix graphs point out that MIRV is uniformly higher than GCC in all categories of instructions (Appendix E), particularly in memory operations. When MIRV produces better code than GCC, it is often because it has reduced the number of 'other' instructions (this happens in go, ijpeg, vortex, and vortex00).

The graphs show that dynamic instruction count is often a very good indication of the number of cycles the benchmark will take to execute. However, there are several counter-examples. For instance, the mirv-O2 instruction count for perl is 2% worse than for GCC-O2 but the binary executes 9.6% faster. The opposite happens on go.

5.2 Comparing SPEC95 to SPEC00

There are several characteristics that differentiate SPEC95 from SPEC00. IPC ranges from 1 to 2 for SPEC95 and 0.6 to 1.8 for SPEC00. The average number of instructions per branch is 4 to 6 for SPEC95 and 4 to 8 for SPEC00 (ignoring ijpeg and the unoptimized binaries).

SPEC00 instruction cache miss rates are very low except for the vortex benchmark. The instruction cache simulated in this work is 16KB. The floating point benchmarks art and equake have very small source code – each is only one source file and have 1270 and 1513 lines of source code, respectively. The integer benchmark mcf is similarly small at 2412 lines of code. These benchmarks are similar to compress, ijpeg, and li95 in the SPEC95 suite. The other SPEC95 benchmarks have a much higher miss ratio than SPEC00. SPEC00 vortex has slightly higher miss rate than SPEC95 version of vortex.

SPEC00 data cache miss rates are much higher than SPEC95. Whereas SPEC95 miss rates are generally less than 2% (5% for compress), SPEC00 miss rates are usually around 4%. art is a particularly notable example with up to a 40% miss rate. Within a given compiler, optimization generally makes the data miss rate worse. This is to be expected as optimizations cause more efficient use of registers, thus eliminating the “easy” load and store operations and leaving those that are essential to the algorithm. A prime example of this is the art benchmark, where the data cache miss rate increases from 15% to 40% as optimizations are enabled from -O0 to -O2. At the same time, however, the number of data references is cut by a factor of three. The low fruit has been harvested and the “essential” memory accesses remain in the benchmark. The unified L2 cache suffers a higher miss rate in SPEC00 as well.

The SPEC00 binaries presented here are much smaller than the binaries for SPEC95. This is one reason that the instruction cache performs so much better for SPEC00. On the other hand, the instruction window is much busier in the SPEC00 than it is in SPEC95 as shown in the register-update-unit utilization graph. One might expect smaller programs to make less usage of the instruction window, but because of the high data cache miss rates it appears that instructions are held up longer in the window.

To summarize the differences between SPEC00 and SPEC95, we saw that IPC and data cache performance were lower for the newer benchmarks, but that these programs exercised the instruction cache less because of their smaller code size. This points out the importance of selecting the appropriate set of benchmarks for a given architectural study. Instruction cache studies should probably avoid many of the SPEC00 benchmarks because they do not stress the instruction cache. On the other hand, data cache studies would emphasize SPEC00 because it strains the data side of the caching system much more than SPEC95. SPEC00 also seems to require a bigger instruction window to avoid window-full stalls. The two suites together seem to provide a nice complement of characteristics; most studies should use both suites.

5.3 Comparing Optimization Characteristics

MIRV and GCC optimizations exhibit similar characteristics across most of the benchmarks but are there exceptions. For example, -O2 optimization usually produces code that runs slightly faster than -O1 code. However, in the case of the vortex benchmark, -O2 code is slightly worse than -O1 code for MIRV. This is due to register promotion which in this case increases the register pressure to the point of introducing additional spilling code.

Branch prediction accuracy is generally much worse for unoptimized binaries. One reason for this is simply the larger number of branches that are executed (20% fewer branches are executed in -O2 than in -O0). For both SPEC95 and SPEC00, prediction accuracies range from roughly 82% to 98% and usually optimizations increase prediction accuracy by 4% or more.

GCC optimizations usually increase the number of instructions retired per cycle (IPC) but for MIRV the opposite is the case.

Both compilers typically demonstrate a reduction in instruction-cache miss rate with optimizations enabled. For vortex, MIRV optimizations also result in an increase in instruction cache miss rate but GCC optimizations actually improve instruction cache performance for this benchmark. For the li benchmark, the reverse occurs.

6. Obtaining and Installing the Binaries

The version 1 binaries used to produce the data in this report are available on the MIRV website [1], including the binaries supplied on the SimpleScalar website [2]. The README file there explains how to install the binaries.

7. Conclusion

This report has introduced the MIRV compiler. As its performance improves, we encourage architecture researchers to use these binaries in conjunction with the SimpleScalar simulation environment as examples of highly optimized programs. As they evolve, these will include advanced optimizations that are not available in GCC and so should be more representative of state-of-the-art compilation techniques.

Acknowledgments

This work was supported by DARPA grant DABT63-97-C-0047. Simulations were performed on computers donated through the Intel Education 2000 Grant.

References

[1] http://www.eecs.umich.edu/mirv

[2] ftp://ftp.cs.wisc.edu/sohi/Code/simplescalar/simplebench.little.tar

[3] Preston Briggs. Register Allocation via Graph Coloring. Rice University, Houston, Texas, USA. Tech. Report. April, 1992.

[4] Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin, Madison Tech. Report. June, 1997.

[5] Standard Performance Evaluation Corporation. SPEC CPU95. http://www.spec.org/osg/cpu95/, Warrenton, Virginia, 1995.

[6] Standard Performance Evaluation Corporation. SPEC CPU2000. http://www.spec.org/osg/

cpu2000/, Warrenton, Virginia, 2000.

[7] http://arch.cs.ucdavis.edu/RAD/gcc-2.7.2.3.ss.tar.gz

[8] http://sourceware.cygnus.com/binutils/

Appendix A. MIRV Optimizations

Frontend		Backend	
Optimize Level	**Filter Applied**	**Optimize Level**	**Filter Applied**
-O2	-fscalReplAggr	-O1	-fpeephole0
-O3	-fcallGraph	-O1	-fpeephole1
-O3	-finline	-O1	-fblockClean
-O3	-ffunctCleaner	-O1	-fcse
-O2	-floopUnroll	-O1	-fcopy_propagation
-O1	-farrayToPointer	-O1	-fconstant_propagation
-O1	-floopInversion	-O1	-fdead_code_elimination
-O1	-fconstantFold	-O1	-fpeephole0
-O1	-fpropagation	-O1	-fpeephole1
-O1	-freassociation	-O1	-fcse
-O1	-fconstantFold	-O1	-fcopy_propagation
-O1	-farithSimplify	-O1	-fconstant_propagation
-O2	-fregPromote	-O1	-fdead_code_elimination
-O1	-fdeadCode	-O1	-fpeephole0
-O1	-floopInduction	-O1	-fpeephole1
-O1	-fLICodeMotion	-O1	-flist_scheduler
-O1	-fCSE	-O0	-freg_alloc
-O1	-fpropagation	-O1	-flist_scheduler_aggressive
-O1	-fCSE	-O1	-fpeephole0
-O1	-farithSimplify	-O1	-fpeephole1
-O1	-fconstantFold	-O1	-fcselocal
-O1	-fpropagation	-O1	-fcopy_propagation
-O4	-fLICodeMotion	-O1	-fdead_code_elimination
-O1	-farithSimplify	-O1	-fpeephole1
-O1	-fconstantFold	-O1	-fblockClean
-O1	-fstrengthReduction	-O1	-fleafopt
-O2	-fscalReplAggr		
-O1	-farithSimplify		
-O1	-fdeadCode		
-O1	-fcleaner		

Table 4. Order of optimization filter application in MIRV. Since the system is based on MIRV-to-MIRV filters, filters can easily be run more than once, as the table shows. The frontend filters operate on the MIRV high-level IR while the backend filters operate on a quad-type low-level IR.

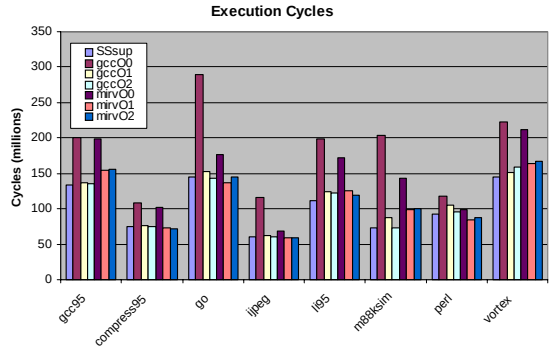
Appendix B. GCC Optimizations

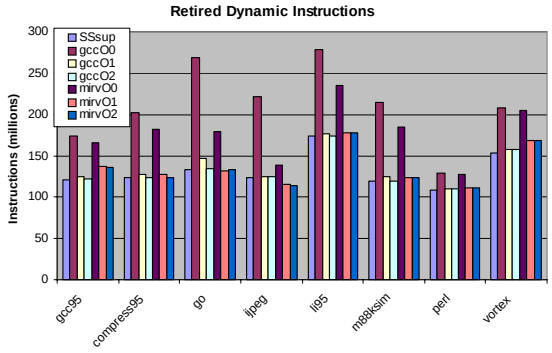
The table shows the optimization sequence when '-O3 -funroll-loops' is turned on. The following flags are enabled: '-fdefer-pop -fomit-frame-pointer -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fthread-jumps -fstrength-reduce -funroll-loops -fpeephole -fforce-mem -ffunction-cse functions -finline -fcaller-saves -fpcc-struct-return -frerun-cse-after-loop -fschedule-insns -fschedule-insns2 -fcommon -fgnu-linker -mgas -mgpOPT -mgpopt'. The table is somewhat incomplete because of the lack of documentation on GCC internal operations.

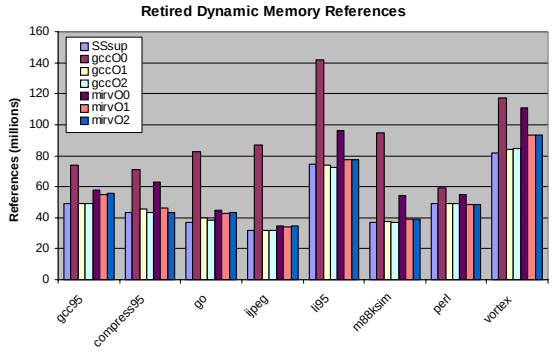
Optimization Applied
jump optimization
cse
jump optimization
loop invariant code motion
strength reduction (induction variables)
loop unroll
cse
coalescing
scheduling (first pass)
register allocation (local, then global)
insert prologue and epilogue code
sheduling (second pass)
branch optimizations (delayed and shortening)
jump opitmization
dead-code elimination

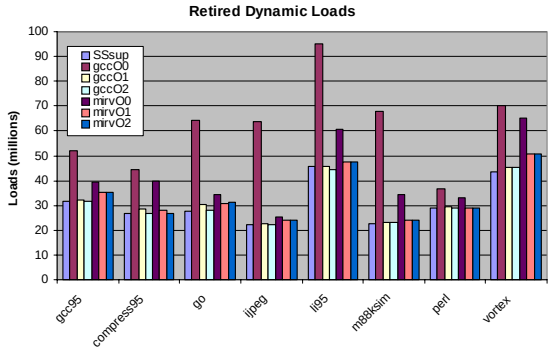
Table 5. Optimization flags in GCC 2.7.2.3/PISA. The GCC compiler is flag based, meaning that an optimization is either on or off. Multiple invocations of an optimization require a special flag (e.g. -frerun-cse-after-loop).

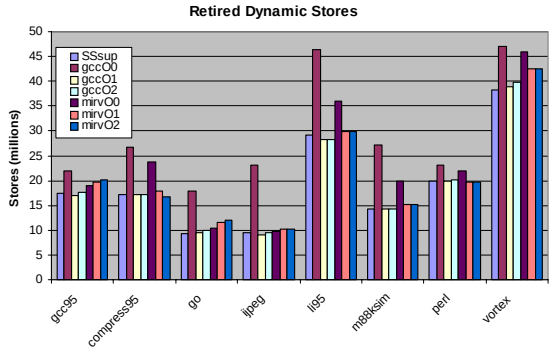
Appendix C. SPEC95 Results

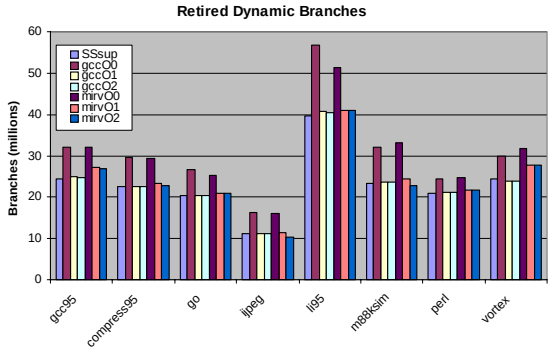


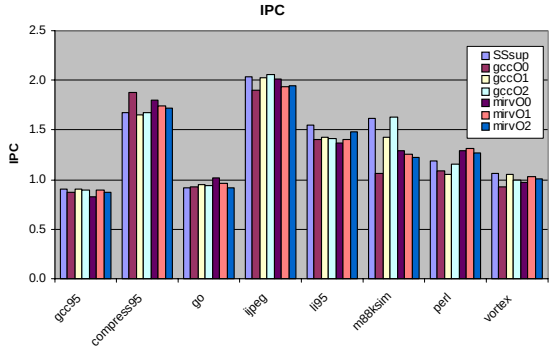


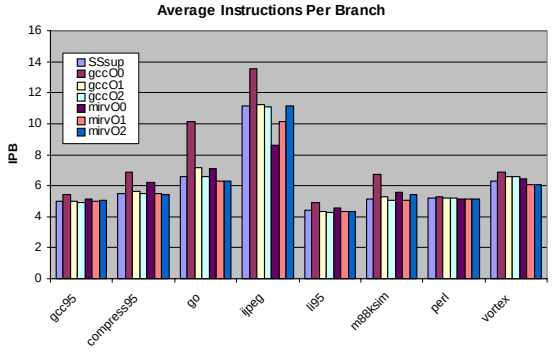


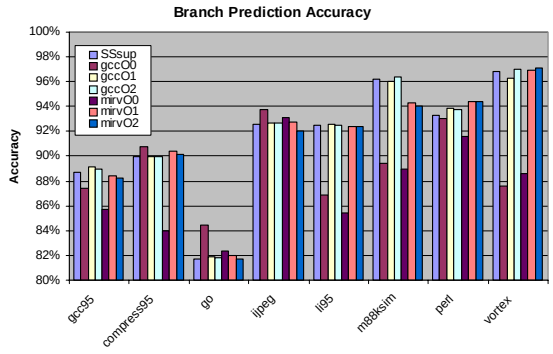


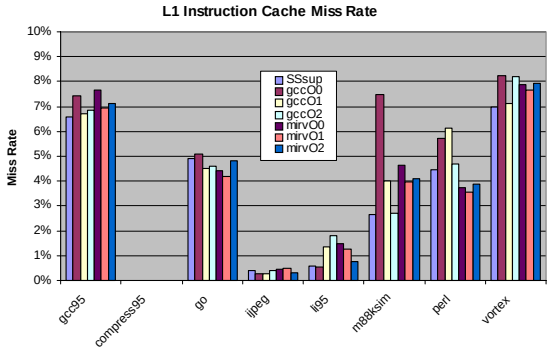


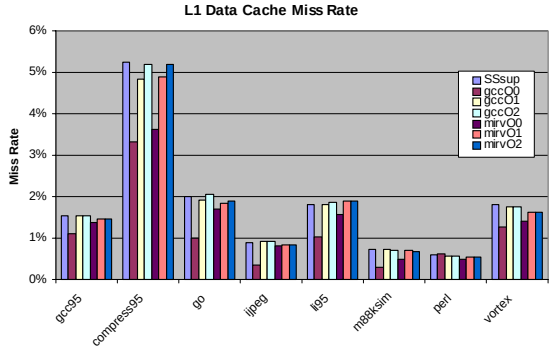


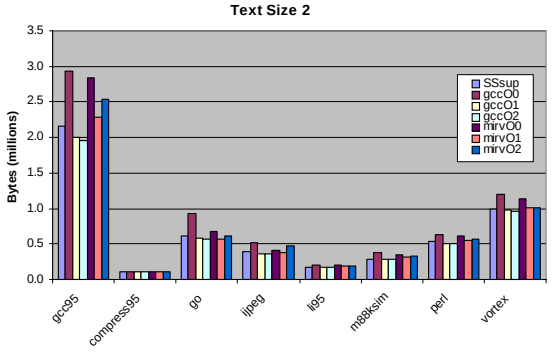


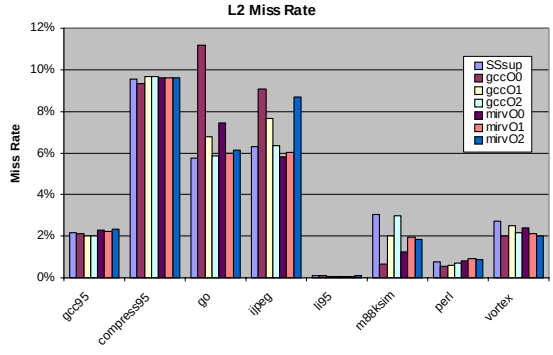


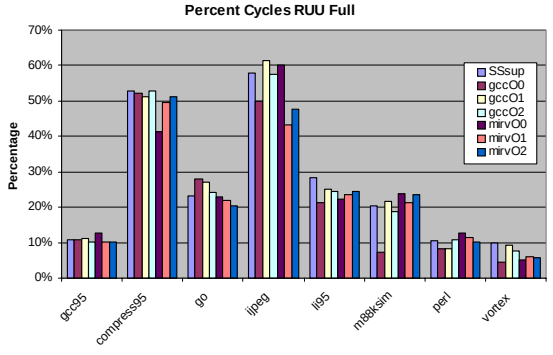

Branch Prediction Accuracy
Accuracy
100%
98%
96%
94%
92%
90%
88%
86%
84%
82%
80%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
gcc95
compress95
go
ijpeg
li95
m88ksim
perl
vortex


L1 Instruction Cache Miss Rate
Miss Rate
10%
9%
8%
7%
6%
5%
4%
3%
2%
1%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
gcc95
compress95
go
ijpeg
li95
m88ksim
perl
vortex


L1 Data Cache Miss Rate
Miss Rate
6%
5%
4%
3%
2%
1%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
gcc95
compress95
go
ijpeg
li95
m88ksim
perl
vortex


Text Size 2
Bytes (millions)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
gcc95
compress95
go
ijpeg
li95
m88ksim
perl
vortex

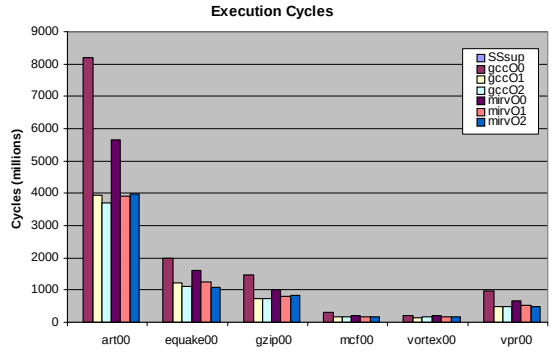

L2 Miss Rate
Miss Rate
12%
10%
8%
6%
4%
2%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
gcc95
compress95
go
ijpeg
li95
m88ksim
perl
vortex

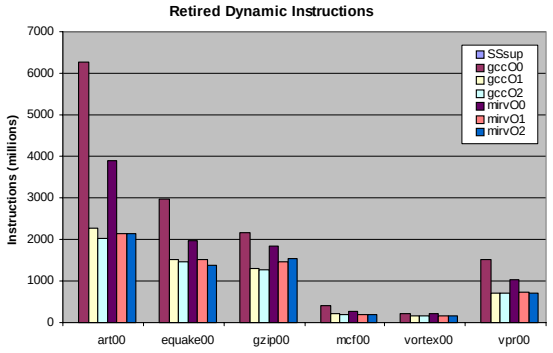

Percent Cycles RUU Full
Percentage
70%
60%
50%
40%
30%
20%
10%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
gcc95
compress95
go
ijpeg
li95
m88ksim
perl
vortex

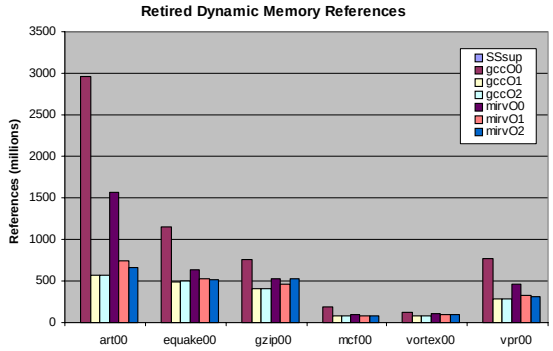
Graph	Special Notes
Execution Cycles	sim_cycle
Retired Dynamic Instructions	sim_num_insn
Retired Dynamic Memory References	sim_num_refs
Retired Dynamic Loads	sim_num_loads
Retired Dynamic Stores	sim_num_stores
Retired Dynamic Branches	sim_num_branches
IPC	sim_IPC
Average Instructions Per Branch	sim_IPB
Branch Prediction Accuracy	bpred_bimod.bpred_dir_rate
L1 Instruction Cache Miss Rate	il1.miss_rate
L1 Data Cache Miss Rate	dl1.miss_rate
Text Size 2	This is computed as bfd_section_size(abfd, sect) of the ".text" section in the binary. This is slightly more accurate than ld_text_size. SimpleScalar instructions are 64-bits each.
L2 Miss Rate	ul2.miss_rate
Percent Cycles RUU Full	ruu_full

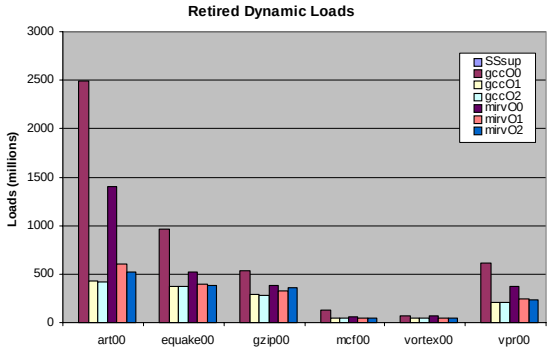
Table 6. Explanation of the graphs in Appendices C and D. Statistics without further explanation are simply the statistic that is produced by the default sim-outorder simulator.

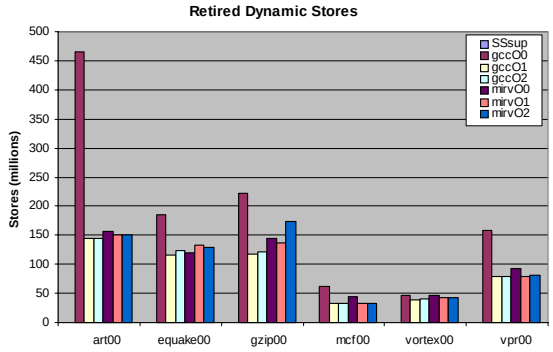
Appendix D. SPEC00 Results

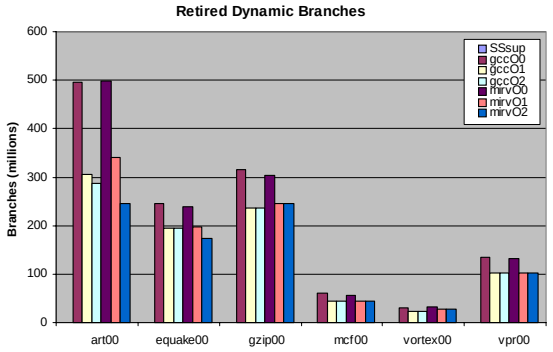


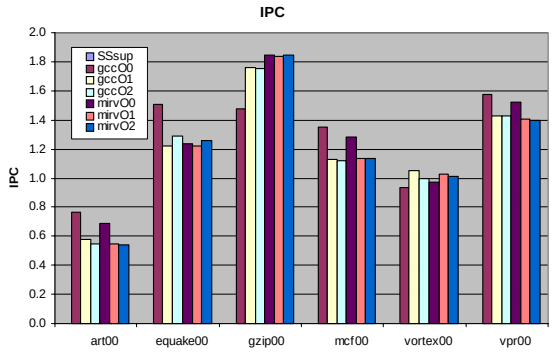


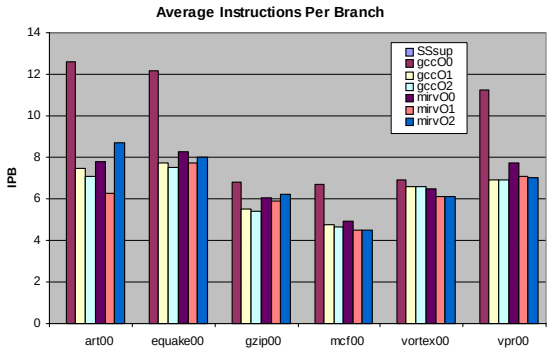


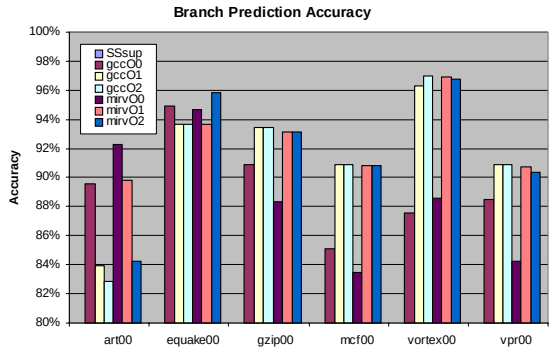


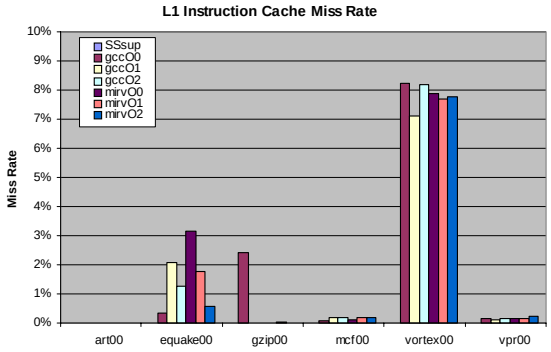


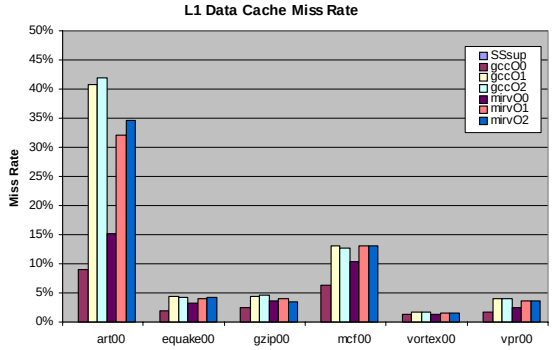


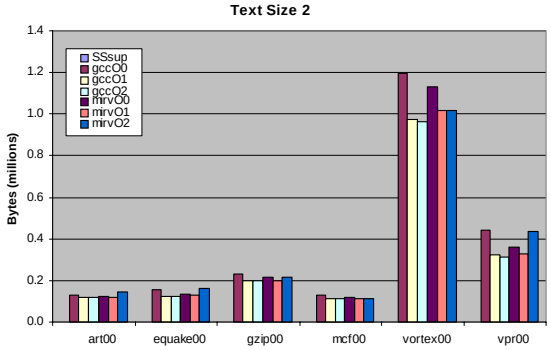


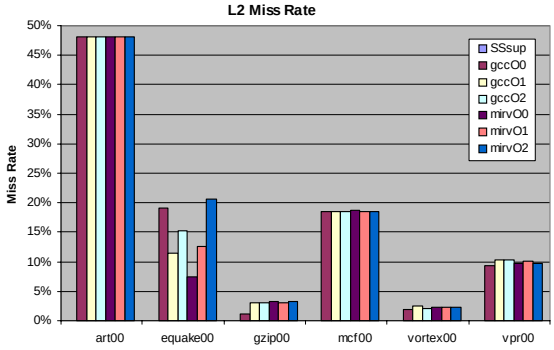


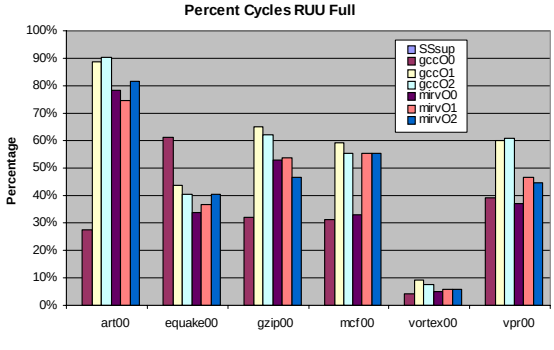

Branch Prediction Accuracy
Accuracy
100%
98%
96%
94%
92%
90%
88%
86%
84%
82%
80%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
art00
equake00
gzip00
mcf00
vortex00
vpr00


L1 Instruction Cache Miss Rate
Miss Rate
10%
9%
8%
7%
6%
5%
4%
3%
2%
1%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
art00
equake00
gzip00
mcf00
vortex00
vpr00

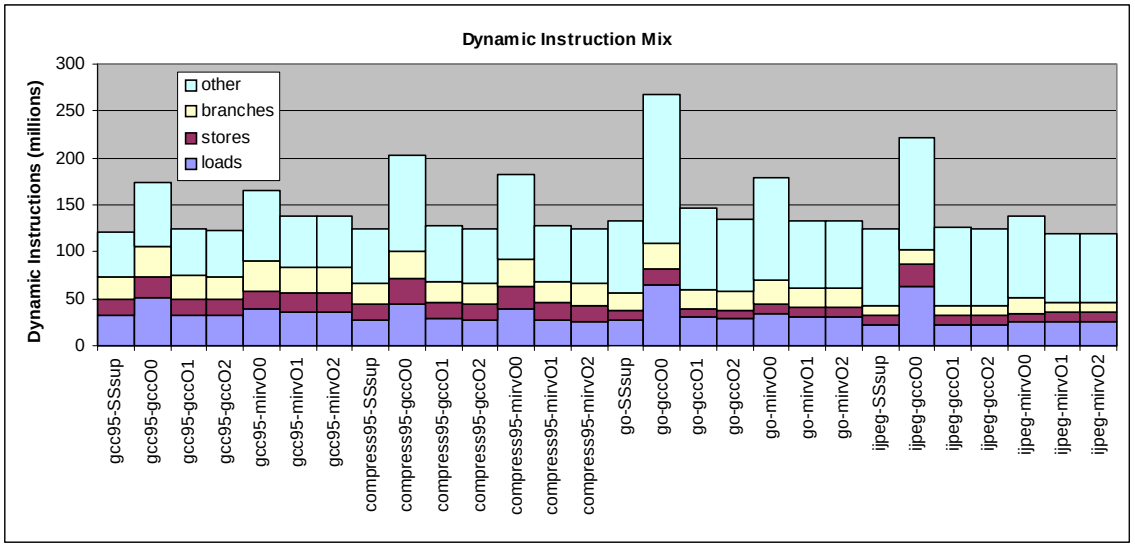

L1 Data Cache Miss Rate
Miss Rate
50%
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
art00
equake00
gzip00
mcf00
vortex00
vpr00

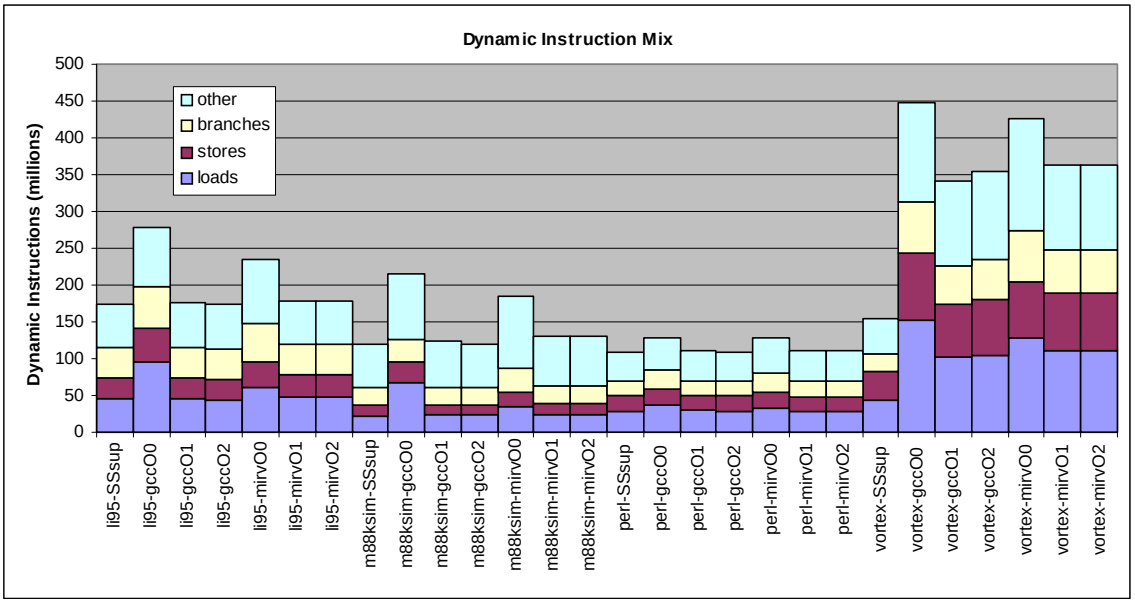

Text Size 2
Bytes (millions)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
art00
equake00
gzip00
mcf00
vortex00
vpr00

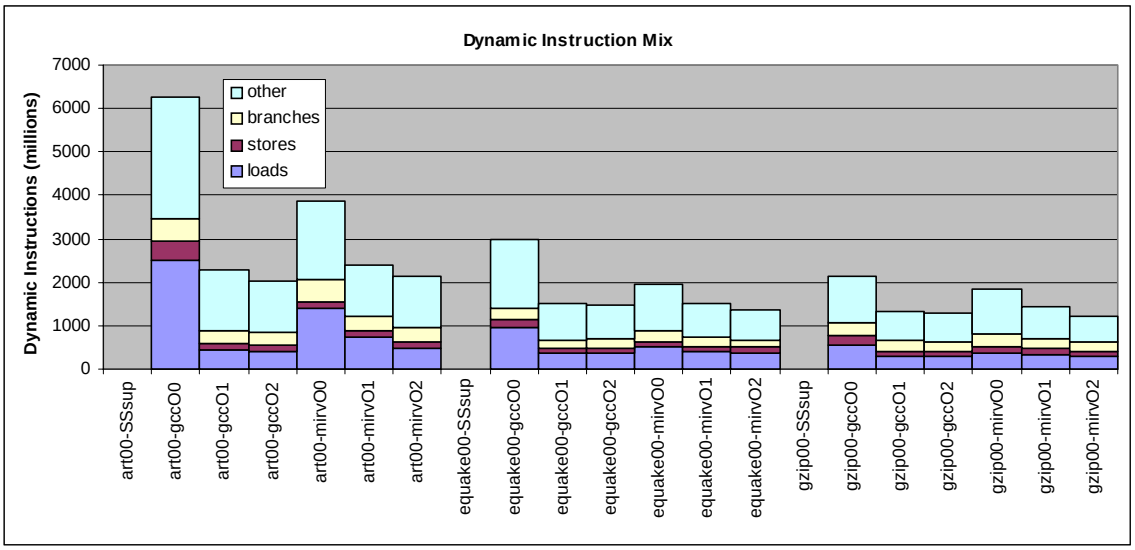

L2 Miss Rate
Miss Rate
50%
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
art00
equake00
gzip00
mcf00
vortex00
vpr00

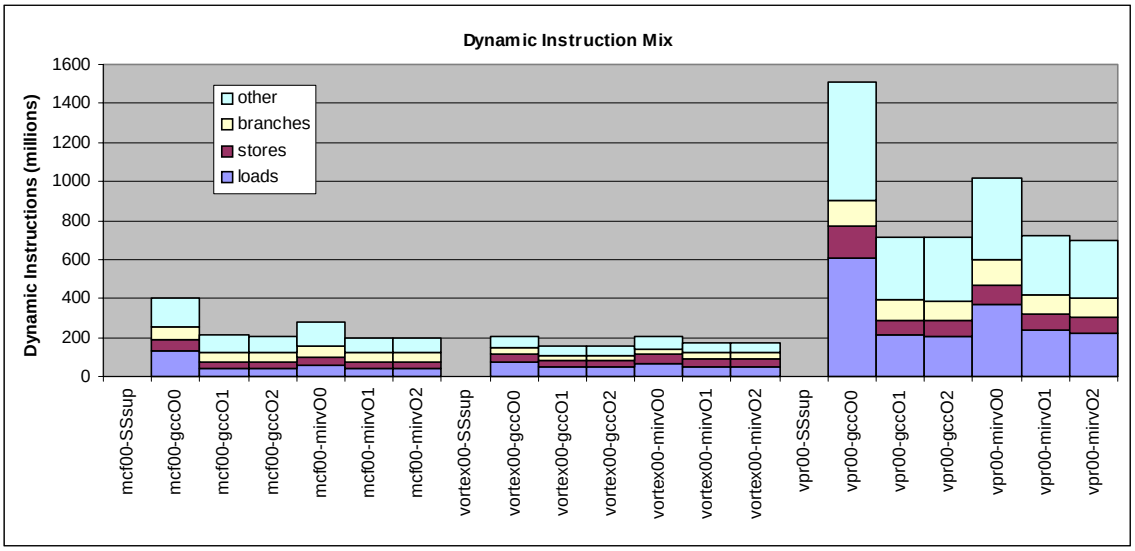

Percent Cycles RUU Full
Percentage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
SSsup
gccO0
gccO1
gccO2
mirvO0
mirvO1
mirvO2
art00
equake00
gzip00
mcf00
vortex00
vpr00

Appendix E. Dynamic Instruction Mix Results






Dynamic Instruction Mix
Dynamic Instructions (millions)
7000
6000
5000
4000
3000
2000
1000
0
other
branches
stores
loads
art00-SSsup
art00-gccO0
art00-gccO1
art00-gccO2
art00-mirvO0
art00-mirvO1
art00-mirvO2
equake00-SSsup
equake00-gccO0
equake00-gccO1
equake00-gccO2
equake00-mirvO0
equake00-mirvO1
equake00-mirvO2
gzip00-SSsup
gzip00-gccO0
gzip00-gccO1
gzip00-gccO2
gzip00-mirvO0
gzip00-mirvO1
gzip00-mirvO2


Dynamic Instruction Mix
Dynamic Instructions (millions)
1600
1400
1200
1000
800
600
400
200
0
other
branches
stores
loads
mcf00-SSsup
mcf00-gccO0
mcf00-gccO1
mcf00-gccO2
mcf00-mirvO0
mcf00-mirvO1
mcf00-mirvO2
vortex00-SSsup
vortex00-gccO0
vortex00-gccO1
vortex00-gccO2
vortex00-mirvO0
vortex00-mirvO1
vortex00-mirvO2
vpr00-SSsup
vpr00-gccO0
vpr00-gccO1
vpr00-gccO2
vpr00-mirvO0
vpr00-mirvO1
vpr00-mirvO2

Appendix F. Detailed Results

Table F.1. Number of execution cycles.

cycles	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	133,198,943	200,481,213	137,146,750	135,925,630	199,353,822	153,676,261	155,570,866
compress95	74,128,332	108,056,529	76,954,251	74,120,871	101,465,965	73,727,346	71,995,564
go	144,963,348	289,772,154	153,516,403	143,099,654	177,020,674	136,487,401	144,615,493
ijpeg	60,867,889	116,708,870	62,016,032	60,362,407	68,882,890	59,562,890	59,092,026
li95	111,990,825	198,753,658	123,987,875	122,404,069	171,851,043	126,223,376	119,688,571
m88ksim	73,873,102	203,182,527	87,329,609	73,359,777	143,770,331	97,886,738	100,992,063
perl	91,785,556	118,283,857	104,616,089	94,904,097	99,226,691	84,818,970	88,070,759
vortex	144,218,250	222,066,885	150,370,197	159,279,949	211,559,470	164,531,061	167,160,492
art00	0	8,183,059,761	3,932,008,214	3,691,442,317	5,664,033,174	3,896,938,836	3,978,797,881
equake00	0	1,983,885,959	1,228,292,079	1,129,167,092	1,592,383,199	1,246,462,154	1,097,503,110
gzip00	0	1,458,535,132	741,395,794	729,502,893	995,964,507	789,611,147	830,910,020
mcf00	0	300,632,079	185,845,520	180,043,497	216,663,192	176,769,988	176,698,141
vortex00	0	222,071,242	150,275,946	159,022,773	211,142,086	165,024,308	166,842,914
vpr00	0	960,620,460	497,678,504	497,481,148	667,190,527	513,963,243	505,800,410

Table F.2. Number of dynamic instructions.

dynInsn	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	121,291,882	173,501,299	124,358,856	122,048,567	165,479,820	136,597,068	135,902,282
compress95	124,007,203	202,848,107	127,320,796	123,989,867	182,370,014	128,062,400	124,117,434
go	132,918,691	268,225,327	146,032,235	134,765,723	179,700,961	131,900,396	132,506,089
ijpeg	123,953,291	221,597,997	125,493,959	124,300,853	138,774,682	114,940,431	114,701,757
li95	173,968,882	277,800,863	176,326,885	173,607,213	234,372,535	177,251,019	177,236,548
m88ksim	119,317,263	214,992,826	124,426,497	119,670,756	184,779,644	122,942,402	123,731,658
perl	108,713,654	129,120,457	110,119,889	109,608,239	127,682,949	111,669,505	111,731,966
vortex	153,682,491	207,233,844	157,864,689	157,918,608	205,467,402	168,991,471	168,981,011
art00	0	6,269,718,143	2,270,057,265	2,024,827,366	3,883,917,755	2,131,438,659	2,137,494,813
equake00	0	2,984,585,045	1,502,806,229	1,459,964,520	1,967,861,216	1,519,585,190	1,382,962,709
gzip00	0	2,149,944,982	1,308,126,550	1,276,220,491	1,840,361,191	1,448,658,876	1,531,669,686
mcf00	0	405,724,021	209,934,857	202,016,959	277,494,737	200,422,476	200,424,704
vortex00	0	207,393,055	158,021,459	158,075,450	205,626,182	169,148,892	169,138,288
vpr00	0	1,510,303,511	710,968,666	710,087,319	1,013,762,258	722,299,249	708,193,105

Table F.3. Number of dynamic memory references.

dynRefs	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	49,334,757	73,873,363	49,352,753	49,361,737	58,200,460	54,892,416	55,461,383
compress95	43,664,405	71,087,446	45,453,686	43,616,577	63,318,786	46,089,721	43,413,349
go	36,716,606	82,177,020	39,763,447	38,071,493	44,760,506	42,363,336	43,109,104
ijpeg	31,856,721	86,688,154	31,571,618	31,708,996	34,770,808	34,330,551	34,393,731
li95	74,677,881	141,546,686	74,055,835	72,648,246	96,516,071	77,473,882	77,450,411
m88ksim	37,052,641	94,852,549	37,360,061	37,214,551	54,389,895	39,072,683	39,001,729
perl	49,025,762	59,632,004	49,364,718	49,074,054	54,951,417	48,596,001	48,677,929
vortex	81,564,028	117,310,095	84,338,484	84,884,654	111,006,867	93,174,770	93,165,111
art00	0	2,953,356,736	572,664,757	562,834,944	1,561,619,480	748,113,677	668,403,628
equake00	0	1,146,908,255	485,738,408	494,960,252	636,607,655	526,424,173	510,722,043
gzip00	0	760,957,701	410,885,840	402,533,982	522,098,333	460,619,365	533,499,622
mcf00	0	191,667,384	77,491,719	77,874,104	100,389,133	76,623,066	76,623,066
vortex00	0	117,424,086	84,450,640	84,996,918	111,120,145	93,287,517	93,277,714
vpr00	0	769,568,893	290,026,962	285,158,550	463,760,059	320,832,075	308,995,984

Table F.4. Number of dynamic load instructions.

loads	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	31,872,008	51,867,054	32,314,881	31,743,080	39,225,924	35,149,405	35,430,480
compress95	26,561,283	44,501,502	28,354,482	26,517,372	39,664,588	28,257,749	26,602,114
go	27,464,121	64,392,437	30,374,262	28,144,373	34,362,737	30,791,030	31,143,981
ijpeg	22,283,318	63,619,861	22,431,106	22,204,793	25,129,726	24,116,600	24,135,665
li95	45,493,244	95,161,631	45,743,456	44,478,428	60,466,443	47,584,005	47,570,254
m88ksim	22,795,190	67,768,522	23,053,841	22,877,056	34,470,734	23,987,331	23,949,875
perl	29,091,584	36,461,407	29,386,498	29,019,370	33,099,127	28,844,092	28,905,023
vortex	43,439,863	70,196,006	45,404,268	45,113,080	65,182,812	50,703,090	50,701,232
art00	0	2,487,176,786	427,202,096	417,372,265	1,405,139,878	598,432,397	518,482,248
equake00	0	960,634,992	369,058,638	371,111,611	517,248,426	393,834,619	381,554,065
gzip00	0	538,118,773	293,914,705	281,592,756	377,580,781	323,275,044	360,690,739
mcf00	0	129,482,527	44,316,226	44,692,447	56,798,038	43,412,222	43,412,222
vortex00	0	70,242,839	45,449,812	45,158,651	65,229,268	50,749,024	50,747,094
vpr00	0	611,078,869	211,472,979	206,577,571	370,396,557	241,442,330	227,526,190

Table F.5. Number of dynamic store instructions.

stores	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	17,462,749	22,006,309	17,037,872	17,618,657	18,974,536	19,743,011	20,030,903
compress95	17,103,122	26,585,944	17,099,204	17,099,205	23,654,198	17,831,972	16,811,235
go	9,252,485	17,784,583	9,389,185	9,927,120	10,397,769	11,572,306	11,965,123
ijpeg	9,573,403	23,068,293	9,140,512	9,504,203	9,641,082	10,213,951	10,258,066
li95	29,184,637	46,385,055	28,312,379	28,169,818	36,049,628	29,889,877	29,880,157
m88ksim	14,257,451	27,084,027	14,306,220	14,337,495	19,919,161	15,085,352	15,051,854
perl	19,934,178	23,170,597	19,978,220	20,054,684	21,852,290	19,751,909	19,772,906
vortex	38,124,165	47,114,089	38,934,216	39,771,574	45,824,055	42,471,680	42,463,879
art00	0	466,179,950	145,462,661	145,462,679	156,479,602	149,681,280	149,921,380
equake00	0	186,273,263	116,679,770	123,848,641	119,359,229	132,589,554	129,167,978
gzip00	0	222,838,928	116,971,135	120,941,226	144,517,552	137,344,321	172,808,883
mcf00	0	62,184,857	33,175,493	33,181,657	43,591,095	33,210,844	33,210,844
vortex00	0	47,181,247	39,000,828	39,838,267	45,890,877	42,538,493	42,530,620
vpr00	0	158,490,024	78,553,983	78,580,979	93,363,502	79,389,745	81,469,794

Table F.6. Number of dynamic branch instructions.

branches	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	24,419,621	32,075,384	24,911,740	24,768,175	31,987,107	27,208,037	26,912,338
compress95	22,449,938	29,619,589	22,447,525	22,447,525	29,375,803	23,225,364	22,761,010
go	20,226,745	26,490,233	20,469,933	20,427,880	25,320,029	20,972,690	20,919,284
ijpeg	11,147,615	16,411,284	11,200,098	11,196,216	16,144,036	11,361,376	10,258,101
li95	39,563,998	56,677,574	40,721,926	40,494,851	51,263,724	41,047,678	41,047,678
m88ksim	23,229,288	32,048,286	23,644,444	23,633,461	33,171,563	24,432,348	22,807,367
perl	20,807,945	24,539,960	21,121,515	21,051,780	24,802,448	21,758,211	21,701,676
vortex	24,386,128	29,994,803	23,970,139	23,940,982	31,815,320	27,718,552	27,718,547
art00	0	496,824,799	305,175,270	286,672,217	497,835,745	340,464,793	245,074,081
equake00	0	245,887,275	194,340,192	194,287,976	237,936,428	196,680,682	172,950,878
gzip00	0	315,190,379	237,097,158	237,096,998	304,468,516	246,300,098	246,142,642
mcf00	0	60,507,881	44,050,368	43,662,048	56,701,362	44,567,160	44,566,440
vortex00	0	30,006,871	23,981,934	23,952,777	31,827,429	27,730,466	27,730,461
vpr00	0	134,367,823	102,717,592	102,668,076	131,409,609	102,297,479	101,097,016

Table F.7. Total number of instructions executed (speculative and non-speculative)

totalinsn	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	142,731,598	197,785,893	146,101,963	143,256,471	191,364,706	160,882,289	159,878,583
compress95	145,313,204	229,548,689	152,356,955	145,305,225	209,208,842	151,114,523	147,488,798
go	164,694,798	302,160,340	178,262,378	166,919,747	213,106,429	164,904,141	164,675,171
ijpeg	132,673,455	230,974,241	134,451,425	133,029,864	147,785,767	123,363,879	123,029,721
li95	212,072,618	320,561,618	214,215,027	207,172,196	277,786,734	225,526,301	225,799,749
m88ksim	127,989,542	222,049,480	132,583,890	128,182,216	194,087,300	139,146,769	140,645,885
perl	124,387,461	141,813,436	124,294,619	125,604,111	143,100,557	125,829,147	125,925,468
vortex	157,204,562	211,428,230	161,684,192	161,403,187	209,951,844	172,990,766	172,832,047
art00	0	6,599,861,689	2,601,964,628	2,358,665,668	4,086,289,721	2,298,462,334	2,417,414,263
equake00	0	3,079,987,564	1,585,588,414	1,541,774,603	2,063,294,135	1,621,353,296	1,432,807,184
gzip00	0	2,373,936,872	1,485,265,785	1,445,983,553	2,055,284,906	1,617,370,836	1,700,150,401
mcf00	0	446,599,029	250,260,853	242,563,503	312,157,139	238,801,643	238,804,394
vortex00	0	211,582,549	161,787,572	161,568,409	210,097,534	173,132,592	173,290,978
vpr00	0	1,633,462,350	825,282,672	823,844,499	1,128,487,567	839,225,111	825,515,677

Table F.8. Total number of memory references executed (speculative and non-speculative).

totalrefs	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	57,644,574	82,971,897	57,441,491	57,468,393	65,405,756	63,848,148	64,271,568
compress95	51,242,289	79,852,930	55,180,437	51,199,467	72,576,616	55,207,752	52,843,584
go	44,987,931	92,231,388	47,675,386	46,637,129	51,533,660	52,652,047	53,144,911
ijpeg	33,743,078	88,824,590	33,280,042	33,610,579	36,390,406	36,239,829	36,373,765
li95	89,978,781	165,736,202	87,894,372	85,331,607	114,404,632	96,624,795	96,718,638
m88ksim	40,537,475	98,421,283	40,590,766	40,650,549	57,198,876	44,358,129	44,540,365
perl	54,572,928	64,639,181	54,224,137	54,952,336	60,183,338	54,290,123	54,119,775
vortex	83,117,340	119,568,855	85,989,966	86,394,959	112,849,860	94,888,095	94,848,785
art00	0	3,122,895,929	626,774,507	640,161,117	1,658,947,251	801,274,286	751,495,150
equake00	0	1,185,730,979	504,513,549	520,254,297	654,951,428	560,047,648	524,313,794
gzip00	0	839,817,109	464,332,540	454,703,182	577,101,119	516,330,530	596,370,658
mcf00	0	210,989,597	89,090,863	91,249,222	109,810,364	88,765,109	88,766,011
vortex00	0	119,689,422	86,104,426	86,506,518	112,958,574	94,990,488	95,078,399
vpr00	0	832,443,022	336,006,141	331,069,790	526,463,967	374,287,250	362,455,251

Table F.9. Instructions per cycle.

IPC	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	0.91	0.87	0.91	0.90	0.83	0.89	0.87
compress95	1.67	1.88	1.65	1.67	1.80	1.74	1.72
go	0.92	0.93	0.95	0.94	1.02	0.97	0.92
ijpeg	2.04	1.90	2.02	2.06	2.01	1.93	1.94
li95	1.55	1.40	1.42	1.42	1.36	1.40	1.48
m88ksim	1.62	1.06	1.42	1.63	1.29	1.26	1.23
perl	1.18	1.09	1.05	1.15	1.29	1.32	1.27
vortex	1.07	0.93	1.05	0.99	0.97	1.03	1.01
art00	0.00	0.77	0.58	0.55	0.69	0.55	0.54
equake00	0.00	1.50	1.22	1.29	1.24	1.22	1.26
gzip00	0.00	1.47	1.76	1.75	1.85	1.83	1.84
mcf00	0.00	1.35	1.13	1.12	1.28	1.13	1.13
vortex00	0.00	0.93	1.05	0.99	0.97	1.03	1.01
vpr00	0.00	1.57	1.43	1.43	1.52	1.41	1.40

Table F.10. Instructions per branch.

IPB	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	4.97	5.41	4.99	4.93	5.17	5.02	5.05
compress95	5.52	6.85	5.67	5.52	6.21	5.51	5.45
go	6.57	10.13	7.13	6.60	7.10	6.29	6.33
ijpeg	11.12	13.50	11.20	11.10	8.60	10.12	11.18
li95	4.40	4.90	4.33	4.29	4.57	4.32	4.32
m88ksim	5.14	6.71	5.26	5.06	5.57	5.03	5.43
perl	5.22	5.26	5.21	5.21	5.15	5.13	5.15
vortex	6.30	6.91	6.59	6.60	6.46	6.10	6.10
art00	0.00	12.62	7.44	7.06	7.80	6.26	8.72
equake00	0.00	12.14	7.73	7.51	8.27	7.73	8.00
gzip00	0.00	6.82	5.52	5.38	6.04	5.88	6.22
mcf00	0.00	6.71	4.77	4.63	4.89	4.50	4.50
vortex00	0.00	6.91	6.59	6.60	6.46	6.10	6.10
vpr00	0.00	11.24	6.92	6.92	7.71	7.06	7.01

Table F.11. Branch prediction accuracy.

BPrate	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	88.70%	87.39%	89.10%	88.96%	85.69%	88.41%	88.22%
compress95	90.00%	90.76%	90.00%	90.00%	83.99%	90.40%	90.16%
go	81.68%	84.42%	81.87%	81.84%	82.31%	81.95%	81.72%
ijpeg	92.61%	93.80%	92.66%	92.66%	93.10%	92.74%	92.02%
li95	92.48%	86.91%	92.58%	92.51%	85.47%	92.40%	92.40%
m88ksim	96.18%	89.45%	95.98%	96.40%	88.97%	94.34%	94.02%
perl	93.34%	93.05%	93.87%	93.77%	91.61%	94.38%	94.38%
vortex	96.80%	87.57%	96.28%	96.98%	88.57%	96.92%	97.12%
art00	0.00%	89.58%	83.95%	82.84%	92.24%	89.77%	84.28%
equake00	0.00%	94.93%	93.65%	93.65%	94.71%	93.69%	95.84%
gzip00	0.00%	90.85%	93.41%	93.41%	88.35%	93.15%	93.14%
mcf00	0.00%	85.12%	90.88%	90.88%	83.51%	90.81%	90.81%
vortex00	0.00%	87.57%	96.27%	96.98%	88.58%	96.92%	96.73%
vpr00	0.00%	88.46%	90.91%	90.90%	84.22%	90.71%	90.31%

Table F.12. L1 instruction-cache miss rate.

IL1miss	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	6.58%	7.41%	6.71%	6.84%	7.65%	6.92%	7.11%
compress95	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%	0.01%
go	4.91%	5.07%	4.50%	4.59%	4.42%	4.19%	4.82%
ijpeg	0.42%	0.29%	0.28%	0.39%	0.46%	0.49%	0.31%
li95	0.58%	0.54%	1.35%	1.80%	1.47%	1.28%	0.77%
m88ksim	2.67%	7.47%	4.03%	2.72%	4.64%	3.98%	4.10%
perl	4.48%	5.70%	6.12%	4.69%	3.72%	3.54%	3.88%
vortex	6.98%	8.23%	7.12%	8.19%	7.89%	7.68%	7.95%
art00	0.00%	0.00%	0.00%	0.00%	0.01%	0.01%	0.00%
equake00	0.00%	0.36%	2.09%	1.26%	3.14%	1.75%	0.57%
gzip00	0.00%	2.41%	0.00%	0.00%	0.01%	0.02%	0.01%
mcf00	0.00%	0.09%	0.19%	0.20%	0.13%	0.19%	0.18%
vortex00	0.00%	8.23%	7.12%	8.19%	7.89%	7.68%	7.77%
vpr00	0.00%	0.15%	0.13%	0.14%	0.16%	0.15%	0.25%

Table F.13. L1 data-cache miss rate.

DL1miss	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	1.54%	1.11%	1.53%	1.54%	1.39%	1.47%	1.46%
compress95	5.23%	3.32%	4.85%	5.19%	3.62%	4.90%	5.18%
go	2.01%	1.00%	1.93%	2.05%	1.71%	1.85%	1.88%
ijpeg	0.90%	0.36%	0.91%	0.91%	0.82%	0.84%	0.84%
li95	1.81%	1.02%	1.82%	1.86%	1.58%	1.88%	1.88%
m88ksim	0.72%	0.31%	0.73%	0.71%	0.50%	0.69%	0.68%
perl	0.60%	0.61%	0.58%	0.58%	0.50%	0.55%	0.55%
vortex	1.81%	1.28%	1.75%	1.76%	1.41%	1.61%	1.62%
art00	0.00%	8.96%	40.73%	41.97%	15.18%	32.12%	34.55%
equake00	0.00%	1.94%	4.38%	4.29%	3.35%	4.05%	4.18%
gzip00	0.00%	2.50%	4.48%	4.54%	3.58%	4.01%	3.47%
mcf00	0.00%	6.35%	12.99%	12.77%	10.31%	13.10%	13.10%
vortex00	0.00%	1.29%	1.76%	1.77%	1.42%	1.62%	1.63%
vpr00	0.00%	1.68%	4.02%	4.07%	2.42%	3.65%	3.71%

Table F.14. Program text size (measurement 1)

textSize	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	2,166,768	2,934,576	2,000,448	1,962,672	2,830,848	2,279,776	2,538,240
compress95	103,840	109,584	105,456	105,264	107,712	105,232	107,552
go	621,600	934,112	581,824	566,400	678,432	561,280	620,144
ijpeg	396,976	520,848	364,752	365,904	414,704	377,280	474,784
li95	180,640	207,792	176,528	176,160	199,536	182,640	183,680
m88ksim	286,864	383,024	289,712	286,608	354,784	308,736	328,192
perl	535,584	627,024	506,992	503,008	621,392	559,184	568,320
vortex	990,928	1,195,328	977,424	966,704	1,132,080	1,017,072	1,017,200
art00	0	131,504	120,384	119,456	123,328	120,384	144,304
equake00	0	154,048	125,904	126,624	134,976	129,232	159,888
gzip00	0	230,448	201,264	200,768	214,720	200,512	213,632
mcf00	0	127,056	114,176	114,400	117,872	115,056	115,600
vortex00	0	1,195,328	977,424	966,704	1,132,080	1,017,072	1,017,152
vpr00	0	439,328	322,768	314,320	363,040	328,336	437,968

Table F.15. Program text size (measurement 2, as described in Table 6).

textSize2	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	2,166,320	2,934,128	2,000,000	1,962,224	2,830,400	2,279,328	2,537,792
compress95	103,392	109,136	105,008	104,816	107,264	104,784	107,104
go	621,152	933,664	581,376	565,952	677,984	560,832	619,696
ijpeg	396,528	520,400	364,304	365,456	414,256	376,832	474,336
li95	180,192	207,344	176,080	175,712	199,088	182,192	183,232
m88ksim	286,416	382,576	289,264	286,160	354,336	308,288	327,744
perl	535,136	626,576	506,544	502,560	620,944	558,736	567,872
vortex	990,480	1,194,880	976,976	966,256	1,131,632	1,016,624	1,016,752
art00	0	131,056	119,936	119,008	122,880	119,936	143,856
equake00	0	153,600	125,456	126,176	134,528	128,784	159,440
gzip00	0	230,000	200,816	200,320	214,272	200,064	213,184
mcf00	0	126,608	113,728	113,952	117,424	114,608	115,152
vortex00	0	1,194,880	976,976	966,256	1,131,632	1,016,624	1,016,704
vpr00	0	438,880	322,320	313,872	362,592	327,888	437,520

Table F.16. Unified L2 miss rate.

UL2miss	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	2.17%	2.14%	2.02%	2.01%	2.28%	2.24%	2.35%
compress95	9.55%	9.36%	9.69%	9.67%	9.60%	9.63%	9.60%
go	5.77%	11.20%	6.79%	5.89%	7.44%	5.97%	6.13%
ijpeg	6.31%	9.06%	7.67%	6.37%	5.82%	6.03%	8.68%
li95	0.10%	0.10%	0.07%	0.06%	0.06%	0.07%	0.09%
m88ksim	3.04%	0.67%	2.01%	3.00%	1.25%	1.93%	1.85%
perl	0.75%	0.54%	0.62%	0.73%	0.80%	0.94%	0.86%
vortex	2.70%	1.99%	2.51%	2.18%	2.41%	2.10%	2.02%
art00	0.00%	48.12%	48.12%	48.12%	48.10%	48.10%	48.12%
equake00	0.00%	19.08%	11.51%	15.20%	7.37%	12.55%	20.65%
gzip00	0.00%	1.08%	3.09%	3.07%	3.32%	3.09%	3.29%
mcf00	0.00%	18.57%	18.52%	18.48%	18.61%	18.48%	18.51%
vortex00	0.00%	1.98%	2.47%	2.11%	2.37%	2.29%	2.38%
vpr00	0.00%	9.42%	10.22%	10.22%	9.72%	10.19%	9.74%

Table F.17. Percentage of time RUU is full.

RUUFull	SSsup	gccO0	gccO1	gccO2	mirvO0	mirvO1	mirvO2
gcc95	10.85%	10.67%	11.22%	10.18%	12.79%	10.27%	10.03%
compress95	52.81%	52.03%	51.37%	52.79%	41.33%	49.48%	51.12%
go	23.26%	27.97%	27.20%	24.06%	22.93%	21.98%	20.38%
ijpeg	58.00%	49.83%	61.33%	57.70%	60.23%	43.31%	47.60%
li95	28.32%	21.41%	25.25%	24.63%	22.34%	23.61%	24.52%
m88ksim	20.21%	7.28%	21.58%	18.90%	23.88%	21.37%	23.52%
perl	10.44%	8.43%	8.19%	10.70%	12.80%	11.60%	10.33%
vortex	9.90%	4.34%	9.19%	7.65%	4.96%	5.90%	5.63%
art00	0.00%	27.55%	88.72%	90.29%	78.48%	74.73%	81.46%
equake00	0.00%	61.10%	43.80%	40.57%	33.83%	36.70%	40.26%
gzip00	0.00%	32.27%	64.87%	62.00%	53.08%	53.88%	46.69%
mcf00	0.00%	31.31%	58.99%	55.55%	32.76%	55.36%	55.40%
vortex00	0.00%	4.30%	9.05%	7.54%	4.84%	5.86%	5.74%
vpr00	0.00%	39.11%	60.04%	61.02%	37.09%	46.48%	44.41%